Fall 9-1-1993

Language Interoperability Issues in the Integration of Heterogeneous Systems ; CU-CS-675-93

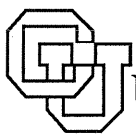
Stanley M. Sutton Jr.
University of Colorado Boulder

Peri Tarr
University of Massachusetts

Language Interoperability Issues in the Integration of Heterogeneous Systems

Stanley M. Sutton, Jr. and Peri Tarr

CU-CS-675-93 September 1993



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Language Interoperability Issues in the Integration of Heterogeneous Systems

Stanley M. Sutton, Jr. and Peri Tarr

CU-CS-675-93 September, 1993

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

Language Interoperability Issues in the Integration of Heterogeneous Systems*

Stanley M. Sutton, Jr.
Department of Computer Science
University of Colorado
Boulder, CO 80309

Peri Tarr
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

Heterogeneity, and consequently interoperability, has become fundamental to large system development and integration. We investigated language interoperability issues in an attempt to integrate two tools written in different languages. We required capabilities such as access to data in both languages, coordination of transactions between languages, and the signaling of events between the languages, among others. These kinds of functionality are typical of advanced heterogeneous applications. We found, however, that current interoperability mechanisms did not provide sufficient support because they tend to focus on a particular domain, e.g., types, events, or transactions. Interoperability between languages depends on the resolution of semantic differences and coordination of functionality in many different domains, such as data, persistence, events and triggers, consistency, and transactions. Interoperability is further complicated by semantic and functional interdependencies within languages.

This paper describes a broad range of functional areas in which we believe typical systems will require interoperability. It discusses issues of interoperability within these areas, describes interactions between them, and proposes solution strategies for supporting the more broad-spectrum interoperability that is required to facilitate large-scale, advanced system integration.

*This material is based upon work sponsored by the Advanced Research Projects Agency under Grants # MDA972-91-J-1009 and # MDA972-91-J-1012. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

1 Introduction

Effective integration of large-scale software systems depends on the ability to accommodate various forms of heterogeneity. For example,

- Software environments must be able to import foreign tools and products and to accommodate the evolution of existing tools, products, and languages [17].
- Software processes require different specification techniques, methodologies, and analysis capabilities, each of which may be useful to different kinds of developers for different tasks [31].
- Megaprogramming depends on the ability to compose large-scale components of existing applications that may have been developed in different languages and with little thought to integration [7].

To accommodate the heterogeneity inherent in these situations it must be possible to combine units, tools, and processes that are written or specified in different languages. The ability to combine components from different languages in turn depends on the interoperability of features in those languages. Language interoperability is consequently fundamental to accommodating heterogeneity in software systems and thus to facilitating software system integration.

Previous work on interoperability has tended to focus on particular aspects of interoperability, such as type model interoperability, event integration, and heterogeneous transactions (e.g., [5, 27, 8]). While the kinds of interoperability support resulting from this work are necessary, they are not sufficient. This is because integrated heterogeneous applications require many kinds of functionality that draw on a wide range of language semantics, many of which interact. Interoperability issues arise across this full spectrum. In order to achieve the desired effects in system integration, it is necessary to understand the semantics of the various features of a language, to account for interdependencies between the features within a language, and to accommodate the differences and interactions between corresponding features in different languages. Language interoperability is thus a broad-spectrum problem requiring broad-spectrum solutions.

To demonstrate the broad-spectrum nature of interoperability concerns, we present a motivating problem based on the integration of two software engineering tools written in different languages. We describe the kinds of functionality that the integrated system must support and identify the semantic areas of the languages on which this functionality depends. Using examples from this integration problem, we describe the semantics of the languages, discuss important differences between them, and illustrate interactions among features within and between the languages. Based on these analyses of differences and interactions, we identify interoperability issues, discuss mechanisms for achieving interoperability, and show how various language features may be used to help support it.

The remainder of this paper is organized as follows. Section 2 presents our motivating problem in system integration, describes the kinds of functionality needed for the integrated system, and suggests the kinds of interoperability required to support this functionality. Section 3 then discusses interoperability issues and opportunities for each of the kinds of interoperability identified, including a review of related work for each area. Finally, Section 4 presents a summary and our conclusions.

2 A Motivating Example: Integrating REBUS and PIC

Our investigation of interoperability in the context of system integration was motivated by an experiment in the Arcadia project [17] to combine two tools that support different phases of software development. REBUS [37] is a process program, written in the software-process programming language APPL/A [34, 36], that supports requirements specification. PIC [42], written using the programming language PLEIADES [38], permits the specification and analysis of modules and their interconnection constraints, and it consequently supports the design phase. To provide a single process that supports both phases of the software life cycle, we attempted to integrate these two systems.

Supporting the requirements and design phases together requires various kinds of interactions between REBUS and PIC. For example, a simple integrated development process, such as the one depicted in Figure 2, might include the following steps. A requirements engineer creates requirements for a new system and enters them into REBUS. When the requirements are sufficiently complete, a design engineer is signaled to initiate the design task. Design elements, created using PIC, must be developed based on requirements stored in REBUS. Consequently, the design engineer must be able to access both REBUS and PIC data concurrently. The need to track dependencies between design elements and requirements (e.g., for traceability) means that the relationship between design modules and the requirements they satisfy must be represented explicitly. Because the REBUS requirements specifications are represented in APPL/A, and the PIC design elements are represented in PLEIADES, the definition of a relationship between requirements and designs spans the two type models. Relationships between requirements and designs must be made persistent for later use by processes that support activities such as coding and testing. The requirements and their corresponding designs must be kept mutually consistent—for example, a change to a requirement may imply the need for a change to the corresponding design element—which means that consistency conditions must be stated and enforced across the two sets of data as a whole. To maintain consistency, the effects of changes to requirements must be propagated to dependent design units; similarly, problems in developing a design may result in changes to the requirements. Multiple engineers and managers must work on both sets of data simultaneously, and repair of consistency violations may require mutually consistent, all-or-nothing modifications to both the requirements and design elements. Thus, concurrency control and atomicity must apply across both sets of data as a whole.

This scenario demonstrates that the integrated REBUS/PIC process must support the full functionality of REBUS and PIC plus several additional capabilities that are not present in either system alone. These include the ability to

- Define and access REBUS-defined and PIC-defined data in a single program
- Define relationships between REBUS requirements elements and PIC design elements
- Make REBUS and PIC data, and the relationships between them, persistent
- Control concurrent access to REBUS and PIC data

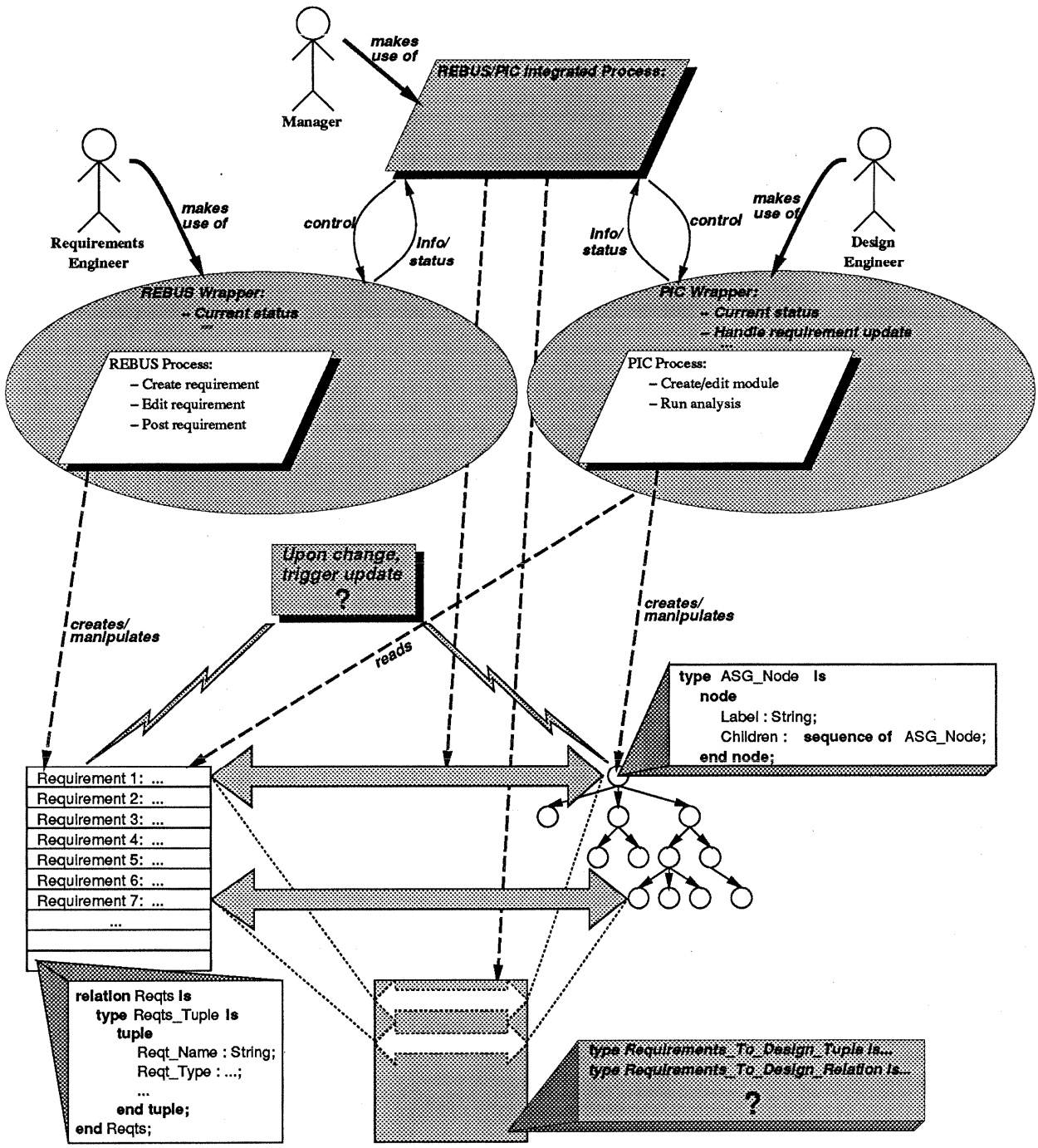


Figure 1: Partial Depiction of the Integrated Requirements/Design Process (existing pieces shown in white; required extensions shown in gray).

- Define and enforce constraints over related REBUS and PIC data
- Query related REBUS and PIC data
- Trigger reactions in the PIC process in response to events in the REBUS process, and vice versa

Thus, to produce the integrated process, it is necessary to achieve interoperability in a number of areas of APPL/A and PLEIADES functionality.

3 Interoperability Issues and Approaches

APPL/A and PLEIADES are both defined as extensions to Ada [40]. APPL/A includes extensions to support software process programming, while PLEIADES has extensions to support software object management. Both languages include support for persistent data, reactive control, consistency management, and transactions, and their models of support are quite different. Consequently, there are many areas in which tools written in the two languages may have to interoperate. The number of areas of interoperability, the semantic differences and interactions between these areas, and limitations on current interoperability support mechanisms, all make achieving the desired interoperability between applications written in APPL/A and PLEIADES a challenge. Further, we believe that many of these interoperability issues are independent of these languages, and that they will arise in other efforts to integrate software systems.

In this section we discuss various semantic and functional areas of the languages in which we found a need for interoperability. For each area, we describe requirements on interoperability, issues involved in achieving it, and potential support mechanisms. We also discuss interactions between the different areas and suggest solution strategies that accommodate them.

3.1 Type Interoperability

The need to relate data defined in different languages is a typical and fundamental problem in system integration. As shown in Figure 2, the dependency of PIC design nodes on REBUS requirements elements must be represented for such purposes as understanding the rationale for a design node and updating the design when requirements change. For this purpose we need to types Requirements_to_Design_Relation and Requirements_to_Design_Tuple. These type definitions must associate instances of APPL/A-defined requirements elements and PLEIADES-defined design nodes, which requires type interoperability.

A mechanism for type interoperability must provide access from one language to types and instances defined in another language. Because it is not possible for a program in one language to directly use types defined in a different language, type interoperability requires the ability to define a "local" type that can be mapped to an external type and used to represent instances of it. Further, it must provide *correct* and *consistent* access to instances of externally-defined types—that is, it must ensure that *all* semantics of the external type are preserved. Consequently, the implementation of "local" type representations must preserve the semantics of the external types,

and it must maintain consistency between the states of external objects and their corresponding "local" implementations.

The ability to create corresponding types and to provide mapping functions between corresponding representations of types defined in different languages has been the subject of a significant amount of research. This research can be classified (as in [41]) as supporting *representation-level interoperability* (RLI) (e.g., [5, 15, 20, 23, 21, 26]) or *specification-level interoperability* (SLI) (e.g., [41, 6, 12]). RLI provides the basic mechanisms for mapping the *physical* representation of an object between its corresponding representations in different languages (e.g., to map an Ada **record** to a C **struct**). SLI enhances RLI with the ability to provide mappings between *abstract data type* (ADT) definitions, and it uses RLI to achieve the physical transfer of objects across language boundaries. Access to objects that are defined externally to a language therefore occurs through an abstract interface that is similar to that of the external ADT. This substantially reduces the problem of preserving the semantics of the original ADT, since ways of manipulating an object that are not part of the ADT's definition can be precluded. This is not the case when RLI is used alone (since having access to just the physical representation for an object may permit erroneous usage of the object). Therefore, we chose an SLI approach to support the accessing of necessary type definitions from other languages. We could therefore define type Requirement_To_Design_Tuple in either APPL/A or PLEIADES; if doAPPL/A, we could use an SLI approach to define an APPL/A ADT that corresponds to the PLEIADES ADT ASG_Node, or conversely, if done in PLEIADES, could define a PLEIADES ADT that corresponds to the APPL/A ADT REBUS_Node.

Satisfying the requirement for consistency between the corresponding representations of an externally defined ADT depends in large part on the semantics of the external ADT. If the external ADT provides *value* semantics (i.e., it does not have a name that is distinct from its state), then a consistency problem will not arise. If it provides *object* semantics (i.e., the object's name is distinct from its state), then a consistency problem between representations in the external language and the importing language may arise if the SLI/RLI mechanism in use maps the entire state of the external object into an internal object. That is, since the object's name is separate from its state, the state may be changed using the external ADT interface after the original state is mapped into its corresponding imported representation. Therefore, providing access to ADTs that are defined in other languages must at minimum accommodate both value and object semantics. In the case of the REBUS/PIC integrated process, both REBUS requirements and PIC design nodes have object semantics.[1] The approach we took to avoid the consistency problem was to map only object names between languages. Access to state is then accomplished through the invocation of the original ADT operations (e.g., through RPC), with any necessary mappings between representations of the object names occurring as part of the implementation of an SLI interface.

Addressing the requirement to preserve the semantics of externally-defined data is partially addressed by SLI. SLI ensures that ADTs will be manipulated in a way that is consistent with

[1]Actually, while all PLEIADES ADTs have object semantics, APPL/A tuples have value semantics, so assignment of requirements tuples will be done by copy rather than by reference. The definition of REBUS requirements includes a unique name field, however, so requirements behave in many situations as objects. We therefore treated them as such in the integrated process.

their operational specification—that is, information hiding can be provided encapsulation can be preserved.

In attempting to use an SLI approach to type interoperability in the REBUS/PIC integrated process, however, we discovered the existence of interactions between type definitions and other important language features. These features include, for example, persistence, reactive control (e.g., triggering), consistency management, concurrency control, and transactions. These other features define various prescribed and proscribed behaviors for ADTs. To fully preserve the semantics of an ADT when it is accessed from another language, the semantics of these other features must also be preserved and must therefore be accommodated by an interoperability mechanism. For example, a language may define access protocols that require instances of an ADT to be manipulated only during a transaction. If this is the case, then the definition of a corresponding ADT in another language must adhere to these protocols. If differences between the access mechanisms in the two languages exist, then they must be resolved before access to the external ADT can be provided. Despite the effect that other language semantics have on the definition and use of types in other languages, we found that these features were not accommodated by any type interoperability mechanism we examined.

In the remainder of this section, we describe the problems we encountered in achieving interoperability in each of these other areas of language functionality. We also describe some of the kinds of interactions that occur between these areas and discuss ways in which these interactions can be accommodated.

3.2 Persistence Interoperability

Once the relationships between requirements and design elements are created, they must be made persistent for later use by the integrated process and other processes (such as analysis and testing). Since these relationships connect instances of types defined in both APPL/A and PLEIADES, and since each language is responsible for the persistence of its own instances, persistence of the relationships requires coordination of the persistence mechanisms in the two languages. A persistence interoperability mechanism must therefore preserve the persistence semantics of the existing ADTs and languages while facilitating the correct implementation of the appropriate persistence semantics for the "heterogeneous" relation and relationship types.

Persistence models typically define either persistence-by-type (e.g., [30, 9, 33]) or persistence-by-instance (e.g., [1, 28]) semantics; APPL/A and PLEIADES are exemplars, where APPL/A defines persistence-by-type semantics over relations while PLEIADES defines persistence-by-instance semantics. Depending on the language in which we choose to define types Requirements_To_Design_Tuple and Requirements_To_Design_Relation, we will need to provide either persistence-by-type or persistence-by-instance semantics. In either case, the artifacts connected by the relationships will have different persistence models, and one of them will have different persistence semantics from those of the relation and relationship types. To provide persistence interoperability, then, it is necessary to overcome the differences between the two persistence models to support the definition of the appropriate persistence semantics for the "heterogeneous" types.

If we define Requirements_To_Design_Tuple and Requirements_To_Design_Relation in APPL/A,

persistence by type applies to instances of these types; thus, it must also apply to the related requirements and designs. The requirements objects are not problematic because they are persistent by type. The PLEIADES-defined design units are persistent by instance, however, so any PLEIADES object that becomes the endpoint of an APPL/A Requirements_To_Design_Tuple instance also be made persistent (if it is not already). An SLI-type interface to the PLEIADES ASG_Node ADT could achieve this effect automatically. Alternatively, an SLI interface to the ASG_Node ADT could provide access to the PLEIADES "make object persistent" and "retrieve persistent object" operations defined on that type for use in APPL/A. However, the persistence-by-type semantics of Requirements_To_Design_Tuple also requires the values of all of its fields to be writable. Thus, object identifiers for design units that are not writable (e.g., in-memory addresses) *cannot* be used as part of an APPL/A definition of type Requirements_To_Design_Tuple. Consequently, an APPL/A type representation of the PLEIADES ASG_Node must not use in-memory object identifiers for the design nodes. Instead, it must correspond to persistent identifiers for the objects that PLEIADES provides, which are writable. An SLI mechanism that makes the PLEIADES-defined ADT ASG_Node available to APPL/A programs must therefore accommodate the PLEIADES persistence-by-instance semantics by ensuring that all design objects are made persistent before they are related to corresponding requirements elements, by providing APPL/A clients with persistent identifiers for design objects, and by performing any necessary translations between in-memory and persistent names for the design objects before invoking PLEIADES operations defined on ASG_Nodes.

If, on the other hand, we define Requirements_To_Design_Tuple and Requirements_To_Design_Relation in PLEIADES, persistence-by-instance will apply to instances of these types. The persistence-by-type semantics of the APPL/A requirements objects is not problematic in this case—instances of requirements will always persist, which means that if they are endpoints of a Requirements_To_Design_Tuple, they will be persistent if required.

The REBUS/PIC interoperability example points out many issues involved in providing persistence interoperability. It also demonstrates the interactions between persistence model interoperability and other language features. With respect to type model interoperability, achieving persistence interoperability potentially affects the way in which an externally defined type is imported (e.g., as an in-memory reference or as a persistent identifier). It may also place additional requirements on the way in which an SLI mechanism implements calls to ADT operations defined in an external language—the SLI mechanism may need to perform translations between internal and persistent representations of types to satisfy any persistence protocols the ADT (or its implementation language) requires. Persistence model interoperability may also depend on transaction model interoperability, since persistence protocols often include a requirement for some sort of transaction to be active before persistent objects can be accessed. In such cases, it may also be necessary for an SLI mechanism to satisfy a transaction protocol on an imported ADT prior to allowing the manipulation of persistent objects.

Related Work:

Interoperability of persistence models has not been addressed explicitly in existing type interoperability support systems. However, since persistence models generally fall into either the

persistence-by-instance or persistence-by-type categories, where persistence-by-instance usually occurs through a "make object persistent"/"retrieve persistent object" protocol, it should often be possible to provide automatic resolution of persistence model differences. Therefore, a broad-spectrum interoperability mechanism should support the specification of an ADT's persistence semantics, and it should be able to use this information to resolve differences between persistence models.

3.3 Reactive Control Interoperability

The need for reactive control in software engineering applications has become apparent over the last several years [27]. More and more systems are using triggers and other reactive control mechanisms to support a range of activities, including metrics collection (e.g., [32]), performance evaluation, and the spawning of processes at key stages of the software lifecycle (e.g., [18, 36, 4]). As systems that employ different reactive control mechanisms are integrated, interoperability support becomes necessary to facilitate the triggering of actions in one underlying component based on events that occur in others. The integrated REBUS/PIC process, for example, may need to listen to events generated by both REBUS and PICto track the progress of the requirements and design processes and to initiate actions at key points during requirements and design development. Thus, we can see that the need to achieve reactive control interoperability requires the ability to recognize and respond to events that occur in other languages.

Recognizing events generated externally is somewhat difficult to accomplish because events in general can only be recognized by the language in which they are defined. Therefore, supporting the recognition of external events requires cooperation between the reactive control mechanisms in the languages in which the events are generated and the ones in which "heterogeneous" triggered actions are written. Propagation of events between languages may occur either *directly* or *indirectly*. Direct propagation occurs when, for example, an APPL/A trigger recognizes an APPL/A event that is of interest to a PLEIADES client directly invokes a PLEIADES action. Conversely, the APPL/A trigger could indirectly propagate the APPL/A event to PLEIADES if, instead of invoking the PLEIADES operation, it notified the PLEIADES broadcast message server that the event had occurred; the broadcast message server would then notify the appropriate PLEIADES clients. These two methods of propagating events from one language to the other have various benefits and drawbacks that are similar to those involved in direct or implicit procedure calling (e.g., [13]). Direct propagation allows the APPL/A trigger to determine whether or not the PLEIADES operation completes successfully (e.g., by examining a return code), which it could not do using indirect propagation. Further, if the APPL/A program had been interoperating with a program written in a language that lacked support for reactive control, direct propagation would be the only way to notify the other system when APPL/A events occurred. On the other hand, direct propagation requires an APPL/A trigger to be modified any time a PLEIADES-supported process is changed such that different entities need to be notified when a given APPL/A event occurs. This would not be the case using indirect propagation—these entities would receive the event signal simply by listening to the broadcast message server. In different applications, direct or indirect propagation may be more or less feasible or appropriate.

Interoperability of reactive control mechanisms may be complicated by interactions with other language features such as transactions and constraints. When atomic (i.e., all-or-nothing) transactions are supported, the preservation of atomicity depends on the ability to control the visibility of event signals and triggered actions—that is, triggering based on the partial, uncommitted results of a transaction should not affect entities outside the scope of the transaction until the transaction has committed. Therefore, a language may either withhold propagation of events out of a transaction until the transaction commits, or it may allow triggered actions to occur and treat their effects as part of the transaction. When event signals are propagated across language boundaries, retaining atomicity may become significantly more difficult, especially if the two languages do not have similar ways of handling this interaction between reactive control and transactions. Thus, in achieving reactive control interoperability using either direct or indirect event propagation, it is necessary to take into consideration any constraints on the propagation of events imposed by both languages' transaction models. Similarly, if reactive control mechanisms can affect the initiation of consistency maintenance actions in another language, then signaling of events that result in these activities may have to take into consideration the required consistency-maintenance semantics of both languages.

Finally, we note that if reactive control is not supported directly within some of the interoperating systems, the only available mechanisms for other languages to identify events that occur within those systems become polling and dispatching. Polling precludes some kinds of trigger semantics (e.g., trigger some action any time a change occurs to an object) because a polled approach cannot recognize the occurrence of all events. Dispatching may allow all necessary events to be recognized, but it comes at the cost of intercepting all operations on ADTs, which requires either the recoding of ADTs (to send messages when operations are invoked) or the recompilation of the other ADT clients with a "wrapper" for that ADT (which performs the message-sending).

Related Work:

Event-based integration mechanisms have been used for a few years as the basis for achieving various kinds of control integration in a number of systems, including Field [27] and Hewlett-Packard's SoftBench [14]. Integration in these systems has been achieved by the use of a single broadcast message server by all integrated tools and by the agreement among these tools on a set of event types. This achieves the effect of control integration, but none of these systems directly supports the interoperability of multiple reactive control mechanisms. More recently, several tool vendors have begun to provide the ability within their tools to translate their tools' events into SoftBench-compatible events (e.g., Amadeus [32] and IDE's Software through Pictures) so that any tool that understands SoftBench events can listen to events from these tools. The existing control integration mechanisms have been limited to simple events with simple messages—data integration has not been present. Further, these control integration systems do not accommodate interactions between reactive control mechanisms and transaction models. Finally, efforts to standardize classes of commonly used events (e.g., CASE Communique) may lead to an event model with which different languages' events can be described and their interoperability supported.

3.4 Consistency Interoperability

Consistency management is the process of keeping objects in states that satisfy some condition for correctness or acceptability. When "heterogeneous" objects are defined, such as the relationships between requirements and design elements, it is often necessary to describe and enforce consistency conditions for these objects, just as it is for software objects in general [2, 35]. In the case of the requirements/design relationships, a number of different consistency conditions may have to be enforced, such as referential integrity over the corresponding requirements and design units. These conditions will apply to all of the relationships in the requirements-to-design relation, and they will be written in terms of the states of *both* the requirements and design units. Since each language enforces consistency definitions only over its own objects, it is necessary to coordinate the consistency management mechanisms in APPL/A and PLEIADES to enforce consistency conditions that span both languages' objects.

The definition of "heterogeneous" consistency conditions requires the ability to refer to objects defined in other languages; the evaluation of these conditions requires the ability access the states of these objects. These aspects of consistency interoperability can be supported through the use of either RLI or SLI. Consistency enforcement, on the other hand, requires the ability to *detect* or *preclude* updates to objects whose states may affect a consistency definition. Detection can be used to support "violate and repair" consistency conditions. For example, one consistency definition on the relationships between requirements and designs might state that a related requirement and design are mutually consistent if the neither has been updated since they were last approved by an engineer. This condition will be violated when either artifact is updated, but these violations should be permitted because the updates are usually necessary. The consistency manager would need to detect the update, however, and respond by sending mail to the person responsible for checking the artifacts so that he/she could approve the changes and thus reestablish consistency. In this case, the detection of a violation enables the enforcement of the desired consistency semantics. Detection in a heterogeneous setting can be achieved using a reactive control interoperability mechanism, which can notify the consistency manager when externally defined objects are updated.

Consistency violation preclusion, on the other hand, requires more than detection of updates. In this case, any actions that would violate a stated consistency definition must be prevented. To prevent consistency violations based on actions that occur in another language, it may be necessary for the other language's consistency manager to enforce consistency conditions across any of its objects whose states affect the "heterogeneous" consistency condition. For example, to enforce a referential integrity constraint on the requirements/design relation, it might be necessary to prevent the destruction of any APPL/A requirements element or of any PLEIADES design element connected by an instance of Requirements_To_Design_Tuple. Destruction of APPL/A objects would have to be prevented through the enforcement of a consistency definition in APPL/A, while PLEIADES object deletion would have to be prevented by a PLEIADES consistency definition. Therefore, the coordinated use of the APPL/A and PLEIADES consistency management systems would be required to achieve the desired effect.

Consistency management interoperability interacts with several other areas of interoperability support. Existing type interoperability mechanisms can facilitate the definition and evaluation

of "heterogeneous" consistency conditions, while the presence of reactive control interoperability can facilitate the enforcement of such definitions. As in the case of reactive control, interactions between consistency management and transaction interoperability may occur if reactions to consistency violations (such as sending mail) could violate atomicity, and these interactions may be especially problematic when consistency repair mechanisms initiate actions across language boundaries. Further, if there are any consistency protocols that must be obeyed to maintain the correct semantics of an ADT (e.g., suspending the enforcement of consistency definitions while operating on the data), these protocols must be satisfied by a type interoperability mechanism.

Related Work:

Interoperability of consistency management mechanisms has not been addressed explicitly in existing type interoperability support systems. Given a specification language for describing the desired effects of heterogeneous consistency definitions, however, we believe it would be possible to automatically generate at least some of the consistency interoperability support required. For example, it might be possible to generate complementary APPL/A and PLEIADES consistency definitions to support the "heterogeneous" referential-integrity constraint.

3.5 Transaction Interoperability

Both APPL/A and PLEIADES provide transaction-related constructs to assure serializable and atomic access across their own objects. The integrated REBUS/PIC process will require comparable access to objects defined in both languages. For example, a manager will need to review both requirements and designs in a stable state; a design engineer may need to read a stable version of the requirements while developing the design; and mutually dependent updates to the requirements and designs may need to be made in an isolated, all-or-nothing manner. Since APPL/A and PLEIADES are each responsible for supporting concurrent, atomic access over their own objects, the kinds of access needed in the integrated process requires coordination of transactions over REBUS and PIC data. More specifically, both coordination of concurrent access and of transaction recovery are needed, which requires interoperability of concurrency-control and recovery mechanisms.[2] We discuss these problems after briefly describing the APPL/A and PLEIADES transaction models.

The PLEIADES transaction model is similar to that of traditional databases in many ways. It provides serializable, atomic access to repositories of persistent data. APPL/A provides transaction-related capabilities through a group of statements that can be used to control access to instances of persistent relation types. These statements offer various combinations of serializability, atomicity, and control over consistency enforcement.

We have examined the problem of coordinating heterogeneous transactions in detail in [39]; we summarize those results here. APPL/A and PLEIADES transactions can be nested in such a

[2]The kinds of coordination indicated here should be sufficient to achieve the effect of simple atomic, serializable transactions over the REBUS and PIC data together. We recognize that such simple transactions are not always sufficient in software engineering applications [3]. However, the mechanisms needed to achieve transaction interoperability in this case are more generally applicable to special design transactions, as demonstrated in [39].

way as to provide concurrent, serializable access to data defined in both languages. This addresses our requirement for coordinated serializable access to heterogeneous data. The coordination of transaction recovery is more problematic, however. The crux of the problem is assuring that two transactions operating in different languages will either both commit or both abort. The problem arises because one of the two transactions must commit (or attempt to commit) before the other. If the first fails to commit, the second can still be aborted. If the first commits, however, and the second one subsequently cannot commit, a non-atomic result will be obtained. This is because there will no longer be any way to revoke the committed effects of the first transaction (i.e., because it releases its locks upon successful commit). Thus, mutual abort cannot be assured because the two transaction management systems operate independently, and because they do not provide sufficient control over functionality related to transaction commit. To address the problem of coordinating recovery, we have recommended extending the languages' transaction models to provide more fine-grained control in this area [39].

Transaction interoperability is affected by the interaction of transactions with features in several other areas. As noted in Section 3.2, transaction interoperability may be a prerequisite to the interoperability of persistent data. As noted in Section 3.3, the behavior of transactions and triggers must also be coordinated, for example, to assure that triggers in one language are not invoked based on results in another language that are subsequently undone. Transactions and consistency management also interact, since the success of a transaction in one language may depend on a consistency condition that involves data from another language.

Related Work:

The heterogeneous transaction problem has been well studied by database researchers (e.g., [8, 10, 24, 25]). Satisfying the requirement for more fine-grained control over recovery mechanisms by exposing two-phase commit operations has been suggested [10, 11], and it has been shown that the absence of fine-grained control over transaction recovery precludes the mutual commit or abort of heterogeneous transactions in the general case [22]. The alternatives to providing fine-grained control are to impose restrictions on the scheduling of local and heterogeneous transactions [24] or to partition the data used by these kinds of transactions [8]. Both of these approaches represent a compromise that may be necessary for system integration if the required control over transaction commit and abort cannot be obtained. None of the prior work in heterogeneous transactions addresses the implications of other kinds of language semantics for transaction interoperability.

3.6 Other Issues

Language interoperability is affected by many more issues than we have addressed here. Some of these arise between APPL/A and PLEIADES, while some do not.

One issue that affects our languages is whether certain kinds of languages features are accessed through language syntax or via an operational interface. One example occurs in the area of transactions. PLEIADES transactions have an operational interface, while APPL/A represents transactions through syntactic statements. Without an operational interface, it is difficult to gain access to a language's features from another—for example, it possible to call PLEIADES transac-

tions from APPL/A, but it is more difficult to do the reverse. In general, the solution to this problem is to put an operational interface on top of the language syntax. It is not always easy to obtain the desired semantics, however. For example, to gain access to APPL/A transactions from within a PLEIADES application, we could encapsulate a whole transaction in a procedure that could be called from PLEIADES, but this approach would preclude some desirable fine-grained interactions between the transaction models. Another approach would be for APPL/A to expose the transaction-management operations that are implicit in the statements; however, assuring the correct use of those operations with respect to nested transactions would then be much more difficult. Data types offer another example of how differences in the way language components are defined affects interoperability. For example, PLEIADES relations are first class-objects, while APPL/A relations are modeled as program units that are not full objects (even in the Ada sense). This has the consequence of limiting (or greatly complicating) the ability to define heterogeneous types involving APPL/A relations.

Two areas that have not been problematic for APPL/A and PLEIADES are control models and signal management (e.g., exception handling). Both languages define unique control-related and signal-related features, but both depend fundamentally on Ada and neither makes extensions that interfere with the Ada mechanisms. There are obvious conflicts between other languages in each of these areas, as between the control models of Ada and Prolog (imperative vs. rule-based), or the run-time systems of Ada and C (each of which typically assumes that it is responsible for signal handling). Interoperability in these and other areas is important but has been beyond the scope of this work.

3.7 Other Related Work

A few studies have addressed broad-spectrum language interoperability, although in contexts different from ours. Ipser et al. have described a multi-formalism specification environment [16]. They believe, as do we, that no single language is ideal for all programs, and that introduction of new languages into closed systems requires considerable reprogramming. Their goal is to design open SDEs that can integrate multiple languages. Their work differs from ours in that it focuses on the integration of domain-specific "formalisms" that are less than complete programming languages. Ipser et al. also provide interpreters for the formalisms, so their work does not address the interoperability of existing language implementations. Language interoperability has also been addressed by Kaiser et al. [19], who have developed an implementation for the Activity Structures process-modeling formalism ([29]) on top of the MARVEL environment kernel. This work focused on clarifying the intended semantics of Activity Structures and in representing those semantics in Marvel. In contrast, we have not been concerned with clarifying the semantics of our languages but with investigating their interaction. Additionally, we have not considered how one language could be used to implement the other; rather, we have been concerned with the interoperation of existing language implementations.

4 Summary and Conclusions

The integration of software systems depends on the ability to combine components written in different programming languages. Language interoperability is thus fundamental to system integration. Languages may define semantics and provide functionality in many different domains, including (among others) data, persistence, events and triggers, consistency, and transactions. System integration may thus require interoperability support in any of these areas.

Achieving language interoperability is difficult because of the differences in language semantic models. Difficulties may also occur because of the functional independence of language implementations (e.g., constraint or transaction models may be semantically similar yet operate independently). Interoperability is further complicated by semantic and functional interactions between features within a language (such as between transactions and triggers or persistence and data models).

Because of possible interdependencies between areas of functionality within a language, interoperability in one area may be necessary to support interoperability in another (as transaction interoperability may be required for persistence interoperability). Even where interoperability in one area may not be necessary for interoperability in another, its presence may still be useful (e.g., trigger interoperability may facilitate heterogeneous consistency management). Conversely, interoperability in one area does not guarantee interoperability in another (as type model interoperability alone does not guarantee persistence interoperability). However, the relationship between different features within a language can sometimes be drawn on to support the integration of features between languages (for example, SLI mechanisms can be used to facilitate both type and persistence interoperability, and events can be used to track potential inter-lingual consistency violations). A thorough understanding of inter-lingual differences and intra-lingual dependencies is thus necessary to fully support the broad-spectrum interoperability that is required to integrate software engineering applications.

Linguistic mechanisms alone are not sufficient for system integration; appropriate design and implementation techniques are also necessary. Nevertheless, interoperability concerns affect and constrain integration architectures and mechanisms. Effective interoperability support is necessary for integration efforts to be most successful. That support can best be provided by treating interoperability as a broad-spectrum problem and developing broad spectrum solutions.

Acknowledgements

We wish to thank the various members of the Arcadia consortium for their comments during discussions of these issues.

5 Referenced Documents

[1] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, Nov 1983.

[2] N. Barghouti. *Concurrency Control in Rule-Based Software Development Environments.* PhD thesis, Columbia University, 1992.

[3] N. Barghouti and G. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, pages 269–317, Sept 1991.

[4] N. Belkhatir, J. Estublier, and M.L. Walcelio. Adele 2: A support to large software development process. In *1st International Conference on the Software Process*, pages 159–170, 1991.

[5] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo, and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13(8), Aug 1987.

[6] L. Blaine and A. Goldberg. Interoperability of abstract values. Technical Note 6, DARPA Module Interconnection Formalism Working Group, Jan 1991.

[7] B. Boehm and W. Scherlis. Megaprogramming. In *Proceedings of the DARPA Software Software Technology Conference*, Apr 1992.

[8] Y. Breitbart, A. Silberschatz, and G.R. Thompson. Reliable transaction management in a multidatabase system. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, 1990.

[9] D. Cohen. *AP5 Manual.* Univ. of Southern California, Information Sciences Institute, Mar 1988.

[10] P.A. Drew. *A la carte: An Implementation of a Toolkit for the Incremental Integration of Heterogeneous Database Management Systems.* PhD thesis, University of Colorado, Boulder, Colorado, 1991.

[11] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

[12] J-C. Franchitti and R. King. Amalgame: A Tool for Creating Interoperating, Persistent, Heterogeneous Components. Technical report, University of Colorado at Boulder, 1992.

[13] D. Garlan and C. Scott. Adding Implicit Invocation to Traditional Programming Languages. In *Proc. 15th International Conference on Software Engineering*, pages 447–455, Baltimore, MD, May 1993. IEEE.

[14] C. Gerety. HP SoftBench: A New Generation of Software Development Tools. Technical Report SESD–89–25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, Nov 1989.

[15] P.B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1), Jan 1987.

[16] E.A. Ipser, Jr., D.S. Wile, and D. Jacobs. A multi-formalism specification environment. In *Proc. 4th ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 94–106, 1990.

[17] R. Kadia. Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project. In *Proc. 5th ACM SIGSOFT Symposium on Software Development Environments*, pages 169–180, Dec 1992.

[18] G.E. Kaiser. Rule-based modeling of the software development process. In *Proc. 4th International Software Process Workshop*, Oct 1988.

[19] G.E. Kaiser, S.S. Popovich, and I.Z. Ben-Shaul. A bi-level language for software process modeling. In *Proc. 15th International Conference on Software Engineering*, pages 132–143, 1993.

[20] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the mercury system. Programming Methodology Group Memo 59–1, Massachusetts Institute of Technology, Laboratory for Computer Science, 1987.

[21] M. Maybee, L.J. Osterweil, and S.D. Sykes. Q: A multi-lingual interprocess communications system for software environment implementation. Technical Report CU-CS-476-90, Computer Science Department, University of Colorado at Boulder, 1990.

[22] J. Mullen, A. Elmargarid, W. Kim, and J. Sharif-Askary. On the impossibility of atomic commitment in multidatabase systems. In *Proc. 1992 Systems Integration Conf.*, 1992.

[23] W.G. Paseman. Architecture of an Integration and Portability Platform. In *Proc. 1988 CompCon*, Mar 1988.

[24] W. Perrizo, J. Rajkumar, and P. Ram. Hydro: A heterogeneous distributed database system. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, 1991.

[25] C. Pu. Superdatabase for composition of heterogeneous databases. In *4th Int'l Conf. on Data Eng.*, pages 548–555, 1988.

[26] J.M. Purtilo. The polylith software bus. Technical Report CS–TR–2469, Department of Computer Science, University of Maryland, 1990.

[27] S.P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57–67, July 1990.

[28] J.E. Richardson and M.J. Carey. Programming Constructs for Database System Implementation in EXODUS. In *Proc. ACM SIGMOD Conference*, 1987.

[29] W.R. Riddle. Activity structure definitions. Technical Report TR 7-52-3, Software Design & Analysis, Mar 1991.

[30] E. Rollins. A simple system for object management in common LISP. In Peter Lee, editor, *Topics in Advanced Language Implementation.* MIT Press, Nov 1990.

[31] H. D. Rombach and M. Verlage. How to assess a software process modeling formalism from a project member's point of view. In *2nd International Conference on the Software Process*, pages 147–159, 1993.

[32] R.W. Selby, A.A. Porter, D.C. Schmidt, and J. Berney. Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development. In *Proc. 13th International Conference on Software Engineering*, pages 288–298, May 1991.

[33] J.M. Smith, S. Fox, and T. Landers. Reference manual for ADAPLEX. Technical Report CCA-83-08, Computer Corporation of America, May 1981.

[34] S.M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming.* PhD thesis, University of Colorado, Boulder, CO, Aug 1990.

[35] S.M. Sutton, Jr. A Flexible Consistency Model for Persistent Data in Software-Process Programming Languages. In *Implementing Persistent Object Bases – Principles and Practice, A. Dearle, G.M. Shaw and S.B. Zdonik, editors*, pages 305–318. Morgan Kaufman, 1991.

[36] S.M. Sutton, Jr., D.M. Heimbigner, and L.J. Osterweil. Language constructs for Managing Change in Process-Centered Environments. In *Proc. 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 206–217, Dec 1990.

[37] S.M. Sutton, Jr., H. Ziv, D. Heimbigner, H. Yessayan, M. Maybee, L.J. Osterweil, and X. Song. Programming a software requirements-specification process. In *Proc. 1st Int'l Conf. on the Software Process*, 1991.

[38] P.L. Tarr and L.A. Clarke. PLEIADES: An Object Management System for Software Engineering Environments. In *ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, Dec 1993.

[39] P.L. Tarr and S.M. Sutton, Jr. Programming Heterogeneous Transactions for Software Development Environments. In *Proc. 15th International Conference on Software Engineering*, pages 358–369, May 1993.

[40] United States Department of Defense, Washington DC. *Reference Manual for the Ada Programming Language*, Jan 1983.

[41] J.C. Wileden, A.L. Wolf, W.R. Rosenblatt, and P.L. Tarr. Specification Level Interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.

[42] A.L. Wolf, L.A. Clarke, and J.C. Wileden. The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process. *IEEE Transactions on Software Engineering*, 15(3):250–263, Mar 1989.